CESUR
Tu Centro Oficial de FP

Automatización y Scripting

En Linux

Índice

1. Comandos Básicos de Linux

Navegación y gestión de archivos:

- **cd**: Cambiar de directorio.
- **pwd**: Mostrar el directorio actual.
- **ls**: Listar el contenido de un directorio.
- **mkdir/rmdir**: Crear o eliminar directorios vacíos.
- **touch**: Crear un archivo vacío.
- **mv**: Mover o renombrar archivos o directorios.
- **cp**: Copiar archivos o directorios.
- **rm**: Eliminar archivos o directorios.

Visualizar y editar contenido:

- **cat**: Mostrar el contenido de un archivo.
- **nano**: Abrir el editor de texto nano.
- **head**: Mostrar las primeras líneas de un archivo.
- **tail**: Mostrar las últimas líneas de un archivo.

Información:

- **man**: Mostrar el manual de un comando.
- **whoami**: Mostrar el usuario actual.
- **groups**: Mostrar grupos a los que pertenece el usuario actual.

Comandos de procesamiento y manipulación de texto:

- **grep**: Buscar patrones en texto.
- **sed**: Editar texto en línea.
- **cut**: Extraer secciones de líneas de texto.
- **sort**: Ordenar líneas de un archivo.
- **uniq**: Mostrar líneas únicas consecutivas.
- **wc**: Contar líneas, palabras o caracteres.

Comandos de permisos y usuarios:

- **chmod**: Cambiar permisos de archivos o directorios.
- **chown**: Cambiar el usuario dueño de un archivo.
- **chgrp**: Cambiar el grupo dueño de un archivo.
- **sudo**: Ejecutar un comando como superusuario.
- **su**: Cambiar a otro usuario.

Comandos varios:

- **echo**: Imprimir texto.
- **clear**: Limpiar la pantalla de la terminal.
- **cmp**: Comparar archivos.
- **read**: Leer entrada del usuario.

2. Directorios especiales

- **. (Punto):**
  - Representa el directorio actual.
  - Útil para ejecutar programas o referenciar archivos dentro del mismo directorio.
  - Ejemplo: ./script.sh ejecuta el archivo script.sh ubicado en el directorio actual.
- **.. (Doble punto):**
  - Representa el directorio padre (el que contiene al actual).
  - Sirve para navegar hacia atrás en la estructura de directorios.
  - Ejemplo: cd .. te mueve al directorio padre del actual.
- **~ (Tilde):**
  - Representa el directorio personal del usuario (también conocido como "home").
  - Útil para acceder rápidamente a archivos o carpetas dentro de tu directorio personal.
  - Ejemplo: cd ~ te lleva a tu carpeta de usuario, como /home/usuario.

3. Creación de un script

1. ¿Qué es un script?¿Qué es un bash?
Un script es un archivo de texto con instrucciones escritas en un lenguaje como Bash, que se ejecutan en el shell para automatizar tareas.

Bash, además de un **shell**, es un **intérprete de comandos**, es decir traduce comandos a instrucciones que ejecuta el sistema operativo.

2. Pasos para crear y ejecutar un script:

1. **Crear el archivo del script:**
   - Usa un editor de texto como nano o touch.
   - Ejemplo: nano mi_script.sh.
2. **Escribir las instrucciones en el script:**
   Empieza con la "shebang" (#!), que indica qué intérprete usar:
   #!/bin/bash
   echo "Hola, este es mi primer script"
3. **Dar permisos de ejecución al script:**
   Usa el comando chmod para permitir que el script se ejecute:
   chmod +x mi_script.sh
4. **Ejecutar el script:**
   Para ejecutarlo desde el directorio actual, usa ./:
   ./mi_script.sh

3. ¿Qué es el PATH?

- El PATH es una **variable** de entorno que almacena **rutas de directorios** donde el sistema **busca programas ejecutables**.
- Si un programa está en un directorio listado en el PATH, puedes ejecutarlo sin especificar su ruta completa.
- Ejemplo: Si el script mi_script.sh está en /usr/bin, ahora puedes ejecutarlo solo escribiendo mi_script.sh en cualquier lugar.

4. Requisitos para ejecutar un programa:

1. El archivo debe ser ejecutable:
    - Usa chmod +x para asegurarte.
2. Debe estar en el PATH o referenciarlo directamente:
    - Desde el directorio actual: ./mi_script.sh.
    - Si está en un directorio de los descritos en PATH, basta con su nombre.

4. Variables en Bash

1. Cómo crear una variable

En Bash, se asigna un valor con = **sin espacios** alrededor del signo igual:

Ejemplo: mi_variable=”Hola, mundo”

Para usar su valor, se coloca $ delante:

Ejemplo: echo $mi_variable

Salida: Hola, mundo

2. Variables de entorno vs. Variables locales

Variables locales: Sólo existen en el script o sesión donde se crearon.

Variables de entorno: Se mantienen en otros procesos.

3. Leer valores del usuario

Se puede utilizar el comando read para pedir entrada al usuario.

Ejemplo:

```
echo “¿Cómo te llamas?"
read nombre
echo “Hola, $nombre”
```

4. Variables especiales
Bash utiliza una serie de variables especiales para manejar argumentos y valores. Los argumentos son datos que se pasan a un script o comando cuando se invocan.

Variable	Contenido
$0	Nombre del Script
$1, $2, $3…	Primer, Segundo, Tercer… argumento pasado al script.
$#	Número de argumentos.
$@	Todos los argumentos

Contenido: script.sh

```
echo $2
echo $#
echo $1
```

Comando: ./script.sh Antonio Berto Carlos
Salida:

```
Berto
2
Antonio
```

5. Comillas y expansión
En Bash, las comillas afectan cómo se interpretan los caracteres especiales y las variables.
Comillas dobles (“”): Interpreta caracteres y variables especiales.

Comillas simples (‘’): Trata todo como texto literal.

Ejemplo:

```
mi_texto="Mundo"
echo "Hola $mi_texto"
echo 'Hola $mi_texto'
```

Salida:

```
Hola Mundo
Hola $mi_texto
```

6. Eliminar variables
Para borrar una variable:

```
unset mi_variable
```

5. Operaciones Aritméticas

1. Uso básico

Para hacer calculso aritméticos en bash utilizamos $(())

Ejemplo:

```
echo $((5 + 3))
```

Salida:

```
8
```

2. Asignar el resultado a una variable

Ejemplo:

```
resultado=$((10 * 2))
echo "El resultado es: $resultado"
```

Salida:

```
El resultado es: 20
```

3. Operadores

Operador	Función
+	Suma
-	Resta
*	Multiplicación
/	División entera
%	Módulo(Resto de división)
**	Exponente(Potencias)

4. Variables en cálculos

Ejemplo:

```
a=10
b=4
echo "Suma: $((a + b))"
echo "Multiplicación: $((a * b))"
```

Salida:

```
Suma: 14
Multiplicación: 40
```

6. El comando test

El comando test se usa para realizar **comparaciones** en Bash. Se puede utilizar de dos formas equivalentes:

Con test directamente:

```
test condición
```

Con [] directamente:

```
[ condición ]
```

Se recomienda utilizar [[]], que es una versión mejorada de test.

1. Comparaciones numéricas

Para comparar números se utilizan los operadores:

Operador	Significado
-eq	Igual
-ne	Distinto
-gt	Mayor que
-lt	Menor que
-ge	Mayor o igual
-le	Menor o igual

Ejemplo:

```
a=10
b=5

if [ "$a" -gt "$b" ]; then
    echo "$a es mayor que $b"
fi
```

Salida:

```
10 es mayor que 5
```

2. Comparaciones de cadenas
Para comparar texto se utilizan los operadores:

Operador	Significado
= o ==	Igualad de cadenas
!=	Diferente
-z	Cadena vacía
-np	Cadena no vacía

Ejemplo:

```
cadena="Hola"

if [ "$cadena" == "Hola" ]; then
    echo "Las cadenas son iguales"
fi
```

Salida:

```
Las cadenas son iguales
```

3. Comparación de archivos
Para condiciones sobre archivos se utilizan los operadores:

Operador	Significado
-e	El existe
-f	Es un archivo regular
-d	Es un directorio
-r	Es legible
-w	Es escribible
-x	Es ejecutable

Ejemplo:

```
if [ -e "archivo.txt" ]; then
        echo "El archivo existe"
fi
```

4. Operadores lógicos
Se pueden utilizar algunos operadores sobre otras condiciones:

Operador	Significado
&&	Operación lógica AND
||	Operación lógica OR
!	Negar condición

Ejemplo 1:
```
if [ "$a" -gt 5 ] && [ "$b" -lt 10 ]; then
    echo "Ambas condiciones son verdaderas"
fi
```

Ejemplo 2:
```
if [ ! -e "archivo.txt" ]; then
    echo "El archivo no existe"
fi
```

7. Condicional if en Bash

El if en Bash permite ejecutar comandos o scripts **según una condición**.

1. Sintaxis básica

```
if [[ condición ]]; then
    # Código a ejecutar si la condición es verdadera
fi
```

2. if-else

```
if [[ condición ]]; then
    # Código si la condición es verdadera
else
    # Código si la condición es falsa
fi
```

3. if-elif-else

Se pueden poner 0 o más condiciones adicionales en bloques elif. Se evalúa cada condición en orden hasta que se cumpla una y se ejecute su código o se llegue al else, que puede haber o no.

```
if [[ condición1 ]]; then
    # Código si condición1 es verdadera
elif [[ condición2 ]]; then
    # Código si condición2 es verdadera
else
    # Código si ninguna condición se cumple
fi
```

8. Bucle while en Bash

El while en Bash ejecuta un bloque de código mientras una condición sea **verdadera**.

1. Sintaxis básica

```
while [[ condición ]]; do
    # Código a ejecutar
done
```

Ejemplo:

```
contador=1
while [[ $contador -le 5 ]]; do
    echo "Iteración: $contador"
    ((contador++))  # Incrementar contador
done
```

Salida:

```
Iteración: 1
Iteración: 2
Iteración: 3
Iteración: 4
Iteración: 5
```

9. Bucle for en Bash

El for en Bash permite **repetir un bloque de código un número determinado de veces**, iterando sobre una lista de valores.

1. Sintaxis básica

```
for variable in lista; do
    # Código a ejecutar en cada iteración
done
```

Ejemplo:

```
for nombre in Juan Pedro María; do
        echo "Hola, $nombre"
done
```

Salida:

```
Hola, Juan
Hola, Pedro
Hola, María
```

2. Bucle for con secuencia de números

Puedes usar {inicio..fin} para generar rangos de números:

```
for i in {1..5}; do
    echo "Número: $i"
done
```

Ejemplo:

```
for i in {1..5}; do
    echo "Número: $i"
done
```

Salida:

```
Número: 1
Número: 2
Número: 3
Número: 4
Número: 5
```

También puedes definir el paso con {inicio..fin..paso}:

Ejemplo:

```
for i in {1..10..2}; do
    echo "Valor: $i"
done
```

Salida:

```
Valor: 1
Valor: 3
Valor: 5
Valor: 7
Valor: 9
```

3. Bucle for con seq

El comando seq genera secuencias de números y se puede usar en bucles for. Puedes usar {inicio..fin} para generar rangos de números:

```
for i in $(seq inicio fin); do
    # Código a ejecutar
done
```

Ejemplo:

```
for i in $(seq 1 5); do
    echo "Número: $i"
done
```

Salida:

```
Número: 1
Número: 2
Número: 3
Número: 4
Número: 5
```

Se puede definir un paso con seq inicio incremento fin:

Ejemplo:

```
for i in $(seq 1 2 10); do
    echo "Valor: $i"
done
```

10. Funciones en Bash

Las **funciones** en Bash permiten reutilizar código y organizar scripts de forma más estructurada.

1. Sintaxis básica

```
nombre_funcion() {
    # Código de la función
}
```

o

```
function nombre_funcion {
    # Código de la función
}
```

2. Llamar a una función

Ejemplo:

```
saludar() {
    echo "Hola, bienvenido a Bash"
}

saludar
```

Salida:

```
Hola, bienvenido a Bash
```

3. Función con parámetros

Ejemplo:

```
sumar() {
    echo "La suma es: $(( $1 + $2 ))"
}

sumar 5 3
```

Salida:

```
La suma es: 8
```

4. Devolver valores

Las funciones pueden devolver números con return, pero solo números (0-255).

Ejemplo:

```
multiplicar() {
    return $(( $1 * $2 ))
}

multiplicar 4 3
echo "Resultado: $?"  # $? captura el valor de retorno
```

Salida:

```
Resultado: 12
```

Tambien se puede devolver texto, usa echo y captura la salida con **echo $( )**

```
obtener_fecha() {
    echo $(date)
}

fecha=$(obtener_fecha)
echo "Hoy es: $fecha"
```

11. Devolver valores en Bash: return, exit y $?

En Bash, las funciones o scripts pueden devolver unos números llamados **códigos de salida** (exit status). Estos normalmente indican si una operación fue exitosa (0) o si hubo un error (1-255).

1. return en funciones

El comando return solo devuelve **números** (entre 0 y 255) dentro de funciones.

2. exit en scripts

exit finaliza completamente un script y devuelve un código de salida.

Ejemplo:

```
#!/bin/bash
echo "Iniciando script"
exit 1  # Termina con código de error 1
echo "Este mensaje no se mostrará"
```

3. $? para capturar códigos de salida

$? Almacena el código de salida del último comando ejecutado.

Ejemplo:

```
ls archivo_inexistente
echo "Código de salida: $?"
```

Si el archivo no existe, ls devuelve 2:

```
ls: no se puede acceder a 'archivo_inexistente': No existe el archivo o el directorio
Código de salida: 2
```

Ejemplo con exit:

```
./script.sh
echo "Código de salida del script: $?"
```

12. Arrays en Bash

En Bash, los **arrays** permiten almacenar múltiples valores en una sola variable. Se acceden por índice y pueden ser manipulados fácilmente.

1. Declara un array
Se usa () para definir un array:

Ejemplo:
```
mi_array=(manzana banana cereza)
```

También puedes declarar un array vacío y agregar elementos después.

2. Acceder a elementos
Se usa el índice (0 es el primero).

Ejemplo:

```
echo "${mi_array[0]}"  # manzana
echo "${mi_array[1]}"  # banana
```

Se puede acceder a elementos desde el final utilizando índices negativos, siendo -1 el ultimo.

Ejemplo:

```
echo "${mi_array[-c]}"  # Banana
```

3. Mostrar todos los elementos
Ejemplo:

```
echo "${mi_array[@]}"
```

Salida:

```
manzana banana cereza
```

Para mostrar los índices:

```
echo "${!mi_array[@]}"
```

Salida:

```
0 1 2
```

4. Añadir elementos
Para agregar un elemento al final del array:

```
mi_array+=(naranja)
```

Para cambiar el valor deuna posición específica:

```
mi_array[2]="sandía"
```

5. Longitud del array

```
echo "Número de elementos: ${#mi_array[@]}"
```

6. Eliminar elementos

Eliminar un elemento:
```
unset mi_array[1]  # Borra 'banana'
```

Eliminar todo el array:
```
unset mi_array
```

7. Recorrer un array con for

Ejemplo:
```
for fruta in "${mi_array[@]}"; do
    echo "Fruta: $fruta"
done
```

13. Automatización en Bash

En Linux, se pueden programar tareas automáticamente con **cron**, **at** o **systemd timers**.

1. cron: Ejecutar tareas periódicamente

cron permite programar tareas repetitivas. Para editar la configuración de tareas programadas, usa:

```
crontab -e
```

Formato de una tarea en cron:

```
* * * * * comando_a_ejecutar
```

Minuto	Hora	Dia	Mes	Dia Semana	Comando
0-59	0-23	1-31	1-12	0-6 (0 = Domingo)	Script o comando a ejecutar

Ejemplos:

Tarea	Efecto
* * * * * comando	Ejecuta el comando cada minuto
0 * * * * comando	Ejecuta el comando al inicio de cada hora
0 12 * * * comando	Ejecuta el comando todos los días a las 12:00
0 0 * * * comando	Ejecuta el comando cada medianoche
0 12 * * 1 comando	Ejecuta el comando cada lunes a las 12:00
0 12 1 * * comando	Ejecuta el comando el día 1 de cada mes a las 12:00

3. at: Ejecutar tareas una sola vez en el futuro

at ejecuta un comando o script en una fecha/hora específica.

Ejemplo: Ejecutar script.sh mañana a las 10 AM:

```
echo "/ruta/del/script.sh" | at 10:00 tomorrow
```

Para ver las tareas programadas:

```
atq
```

3. systemd timers: Alternativa moderna a cron

Si tu sistema utiliza **systemd** (normalmente lo hace), puedes programar tareas con systemd timers, que son más flexibles que cron. Se configuran con archivos .service y .timer.